HUMBOLDT-UNIVERSITÄT ZU BERLIN

Bachelorarbeit

Ein generischer Ansatz zur Implementation benutzerdefinierter Statistik-Applikationen

A generic approach to implement user-defined statistics applications

Axel Hylla
Matrikelnummer 559874
Bachelor of Science Volkswirtschaftslehre
Email: hu.axel@hylla.de

29.01.2017

Humboldt-Universität zu Berlin
Wirtschaftswissenschaftliche Fakultät
Ladislaus von Bortkiewicz Chair of Statistics
Erstprüfer: Prof. Dr. Wolfgang K. Härdle
Zweitprüfer: Prof. Dr. Stefan Lessmann
Betreuer: Dr. Sigbert Klinke

Zusammenfassung

Die vorliegende Arbeit stellt eine Implementation eines generischen Algorithmus zur Erzeugung von Statistik-Applikationen vor, die auf dem Paket “shiny” der Programmiersprache “R” beruhen. Häufig benötigte Programmcode-Lösungen solcher Shiny-Apps werden durch reduzierte Deklarationen im Algorithmusaufruf vereinfacht. Darüber hinaus werden für einige häufig gebrauchte Eingaben statistischer Funktionsparameter standardisierte Macro-Funktionen angeboten.

Abstract

This paper presents an implementation of a generic algorithm for the creation of statistics applications based on the ”shiny” package of the programming language ”R”. Code solutions within such shiny apps that are often required are simplified by declarations of reduced length. Furthermore, some regularly used parameter inputs of statistics functions are converted to standardized macros.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

1.1 Aufgabenstellung und Motivation

Die vorliegende Arbeit folgt in ihrer Umsetzung der Idee meines Betreuers, Dr. Sigbert Klinke. Der Ladislaus-von-Bortkiewicz-Lehrstuhl für Statistik hat sich in den vergangenen Jahren einen Namen auf dem Gebiet der Elekronisierung von Lerninhalten, statistischer Algorithmen und Abhandlungen erarbeitet. Hierzu gehört auch die Entwicklung so genannter Quantlets - kleiner Programme, die auf der Statistiksoftware "R" und dem dazugehörigen Paket "shiny" basieren. Eine Auswahl davon ist unter dem Namen MM*Stat online abrufbar (s. `https://www.mm-stat.org`) und beruht auf der Abhandlung "Introduction to Statistics, Using Interactive MM*Stat Elements"[1], die mit zahlreichen Beispielen interaktiver statistischer Programmen aufwartet.

Die Erstellung dieser Programme erfordert zum einen die genaue Kenntnis der inneren Funktionsweise der "R"-Programmiersprache als auch des "shiny"-Programmpaketes für "R", zum anderen die Bewältigung immer wieder neuer Herausforderungen bei der Abstimmung der verschiedenen Eingabefelder (Inputs) untereinander und in Bezug auf die Ausgabefelder (Outputs). Hieraus entstand die Idee, ein Werkzeug zu entwickeln, das den Zugang zu diesen Aufgaben vereinfacht und für einige wiederkehrende Herausforderungen Standardlösungen bereit stellt. Auf diese Weise sollen Statistik-Anwender, darunter vor allem auch Lernende, weniger Hemmungen haben, eigene Statistik-Applikationen zu entwickeln. Die Aufgabe bestand also darin, ein Programm zur Generierung von Programmen zu erstellen.

Hierzu möchte ich ausführen, dass ich während meiner circa dreizehnjährigen Anstellung als Programmierer bei zwei Computerspielefirmen durchaus schon ähnliche Aufgaben umsetzen durfte, indem zum Beispiel ein Parser für die Ablauf-Scripte cinematographischer Zwischensequenzen oder ein grafisches Werkzeug zur Erstellung so genannter Behavior-Trees für künstiche Intelligenz angefordert wurden. Aus diesem Grund freute ich mich auf die mir hier gestellte Aufgabe, da ich mich erinnerte, wie herausfordernd und zugleich beglückend die Umsetzung derartiger Ideen sein kann.

1.2 Ansatz und Umfang der Arbeit

Ein wichtiger Punkt für ein solches Werkzeug ist die Frage des Spektrums der geplanten Anwender. Es durfte vorausgesetzt werden, dass sich die Anwender bereits grundsätzlich mit "R" und auch der Funktionsweise von "shiny" beschäftigt haben und wissen, wie Input- und Output-Elemente erstellt und mit Leben erfüllt werden. Es ging also eher darum, übliche Hindernisse bei der Erstellung von shiny-Apps zu umschiffen (mehr dazu im Abschnitt 2.2 "shiny").

Ebenso wurde klar definiert, welcher Art und welchen Umgangs die shiny-Apps sein werden, die mit dem geplanten Werkzeug erstellbar sein würden, nämlich genau solche, wie sie in o.g. Lehrbuch als interaktive Elemente vorgestellt werden. Ich konnte

[1] siehe **Härdle, Klinke und Rönz (2015)**[1]

diese Applikationen also als Leitplanken meiner Entwicklung betrachten und verwenden. Wenn mein Programm in der Lage sein würde, sie der Funktionsweise und dem Erscheinungsbild nach zu replizieren, so würde ich das als Erfolg betrachten.

Somit waren der Einspringpunkt definiert, aber auch eine horizontale wie vertikale Begrenzung des programmiertechnischen Teils der Aufgabenstellung festgelegt. Im schriftlichen Teil sollte das Vorgehen dokumentiert und begründet werden, wobei ich festhalten muss, dass circa 80-90% der Arbeitsleistung in die Erstellung des Programmcodes fließen mussten, der sich durchaus als eine Herausforderung erweisen sollte, die ich anfangs etwas unterschätzt hatte.

2 Methodik

Der Umsetzung der Programmierung muss eine Analyse der verwendeten Programme und Pakete sowie der zu replizierenden interaktiven Elemente vorausgehen, um sich für den besten Ansatz entscheiden zu können.

2.1 R

"R" ist eine in den Neunzigern des vorigen Jahrhundertes entwickelte Programmiersprache mit einem klaren Fokus auf statistische Problemstellungen. Aufgrund ihrer Flexibilität und dem bequemen Zugang zu Statistik-typischen Datenstrukturen ist sie weiterhin auf dem Vormarsch. Während meines Studiums an der Wirtschaftswissenschaftlichen Fakultät konnte ich beobachten, wie mehr und mehr Studenten, aber auch Lehrstühle, auf diese Programmiersprache umstiegen, so zum Beispiel der Lehrstuhl für Ökonometrie, der zuvor in den Bachelorkursen das Programm eViews verwendete.

Die hohe Flexibilität und das immense Spektrum an verfügbaren zusätzlichen Paketen hat aber auch Nachteile. Die Syntax ist abwechslungsreich und es gibt keine allgemein anerkannten Konventionen für die Benennung von Variablen, Funktionen und Parametern. Auch die Art und Weise, wie Funktionsparameter interpretiert werden, kann sich von Paket zu Paket unterscheiden.

Die innere Struktur bietet bei der Evaluation der Programmier-Ausdrücke immer wieder Überraschung. So musste ich während der Arbeit an diesem Projekt mehrfach schmerzvoll erfahren, dass die Unterschiede zwischen einer Funktion, einem Call, einer Expression, einer "language" und einem Quote zwar fein, in ihren Auswirkungen aber erheblich sein können, was insbesondere bei der Konvertierung der einen zu der anderen Form viel Obacht und kreative Ideen erforderte.

Ich erkannte schnell, dass es aufgrund der vielfältigen Möglichkeiten eines Funktionsaufrufs in "R" schwierig werden würde, alle denkbaren Varianten von Parameter-Übergaben korrekt abfangen zu können, und ich bin mir sicher, dass es auch beim vorliegenden Ergebnis Möglichkeiten gibt, das Programm dazu zu bringen fehlzuschlagen.

Dennoch entschied ich mich für die Umsetzung in Form eines einzigen Funktionsaufrufs, dessen Argumente die simulierten Calls beinhalten, die ansonsten für die Gene-

rierung von shiny-Input- und Output-Elementen verwendet würden. Eine Alternative wäre die Implementation von Klassen gewesen, die dann mit importierten Daten gefüttert und so zur Ausbildung einer shiny-App geronnen wären. Dieser Ansatz böte eine deutlich umfangreichere Abbildung aller denkbaren shiny-App-Varianten, hätte den Rahmen der vorliegenden Arbeit jedoch bei weitem gesprengt. Ich musste mich daher für eine Beschränkung der Funktionalität meiner Lösung entscheiden.

2.2 shiny

Das R-Paket `shiny` und die Erweiterungen `shinydashboard` and `shinyBS` dienen der Erzeugung von HTML-basierten Applikationen, die anschließend entweder in einem von R-Studio erzeugten Fenster oder in einer Browserumgebung verwendet werden können. Eine auf einem Server abgelegte App kann zeitgleich von mehreren Nutzern aus online aufgerufen und gestartet werden. Somit ist `shiny` bereits ein mächtiges Werkzeug für Statistik-Applikationen.

Für eine Programmschnittstelle, die ein User-Interface erzeugt, verfügt `shiny` jedoch über einige ungewöhnliche Eigenschaften und Limitierungen. Die Steuerung des Informationsflusses von den Inputs zum Output gestaltet sich oft als schwierig und unvorhersehbar.

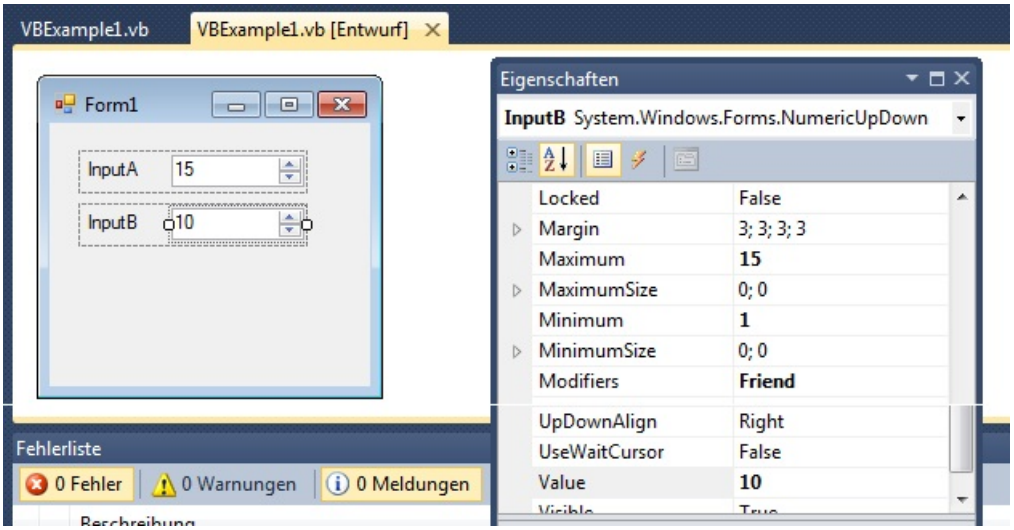
2.2.1 Vergleich des Signalflusses von shiny mit VB.net

Die üblichen Anwendungen zur Erzeugung von User-Interfaces arbeiten mehrheitlich mit einem Informationsfluss-System, das am Beispiel der Programmiersprache VB.net, die eine besonders einfache und anschauliche Methode zum Design und Verdrahtung von UI-Elementen bietet, erläutert werden soll. Abb. 1 zeigt ein sehr simples UI-Element, auch Form genannt, mit nur zwei Inputs, nämlich zwei numerischen Inputs mit Minimum- und Maximumwerten. Letztere können, unter vielen weiteren Eigenschaften, im Eigenschaften-Fenster des jeweiligen Elements eingestellt werden.

Der Signalfluss jeglicher User-Interaktion dieser Elemente wird in Form von Callbacks ermöglicht, die unmittelbar auf die Benutzeraktion reagieren. Dabei ist es unerheblich, ob die Reaktion auf andere Input-Elemente zurückwirken soll, oder weitergehende Konsequenzen, z. B. auf einen Output haben soll. Nehmen wir an, der Wert von Input A soll stets auch der maximal mögliche Wert des Inputs B sein. Ein Doppelklick auf das Element im Designer-Modus öffnet das dazugehörige VB-Script und erzeugt dort eine Callback-Funktion, deren Code zur Ausführung kommt, sobald der Nutzer die naheliegendste Aktion in Bezug auf das UI-Element ausführt (in diesem Fall, den Wert zu ändern).

Nehmen wir weiter an, dass letztlich eine Funktion `Redraw()` ausgeführt werden soll, die die beiden Werte der Inputs A und B verwendet. Der Beispiel-Code 1 zeigt, wie der Programmierer volle Kontrolle über die Reihenfolge des Anpassens des Maximums und des Werts von B gegenüber dem Ausführen weiteren Codes hat.

Abbildung 1: VB.net-Beispiel 1 Designer



Programm-Code 1: VBExample1.vb

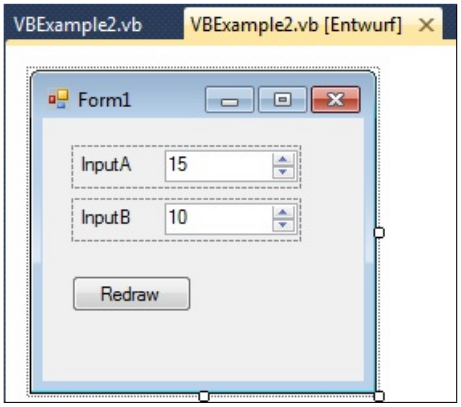
```
Public Sub Redraw()
    ' Do Something ...
End Sub

Private Sub InputA_ValueChanged(sender As System.Object, e As ↙
    ↳ System.EventArgs) Handles InputA.ValueChanged
    InputB.Maximum = InputA.Value
    InputB.Value = Math.Min(InputA.Value, InputB.Value)

    ' call Redraw after all input values are correctly set
    Redraw()
End Sub
```

Noch eindeutiger und transparenter für den Anwender geschähe dies im zweiten Beispiel (Abb. 2 und Code 2), indem ein gesonderter Button zum Ausführen des Redraw-Befehls hinzugefügt wird, der zwingend nicht gleichzeitig mit der Veränderung des Input-B-Wertes aktiviert werden kann. Dabei jedoch verliert die Anwendung an Interaktivität, wenn wir uns an die Stelle des Inputs einen der shiny-typischen Slider denken, und der neugezeichnete Plot-Ausdruck auf jede Verschiebung des Sliders unmittelbar reagieren soll.

Abbildung 2: VB.net-Beispiel 2 Designer



Programm-Code 2: VBExample2.vb

```
Public Sub Redraw()
    ' Do Something ...
End Sub

Private Sub InputA_ValueChanged(sender As System.Object, e As ↙
    ↳ System.EventArgs) Handles InputA.ValueChanged
    InputB.Maximum = InputA.Value
    InputB.Value = Math.Min(InputA.Value, InputB.Value)
End Sub

Private Sub Btn_Redraw_Click(sender As System.Object, e As ↙
    ↳ System.EventArgs) Handles Btn_Redraw.Click
    ' call Redraw explicitely
    Redraw()
End Sub
```

Das Paket `shiny` verfolgt hingegen einen ganz anderen Ansatz, indem es versucht, automatisch zu ergründen, welche Input-Werte von welchen Output-Elementen benötigt werden. Der den Outputs zugeordnete Code holt sich die Input-Werte also bei Bedarf. Diese Arbeit liefert die shiny-interne so genannte Reactive-Engine, die alle Output-Elemente (und auch zwischengeschaltete Reactive-Values), die einen Input-Wert verwendet haben (entweder bei der Initiierung oder danach), mit einem Invalidation-Flag markiert, sobald einer dieser Input-Werte sich geändert hat (Hinweis: Der "Wert" eines Buttons ändert sich mit jedem Klick, indem intern ein Integer um 1 erhöht wird), was die erneute Evaluation des Output-Codes zur Folge hat.

Hierdurch ergeben sich zwei Probleme: Zum einen ist die Reihenfolge, in der Outputs invalidiert und damit erneut ausgeführt werden, wenn mehrere Outputs von dem- oder denselben Inputs abhängig ist, nicht bestimmbar. Meine Tests mit der Anordnung der Elemente im Code erbrachten diesbezüglich keine eindeutigen Ergebnisse. Zum anderen ist die Implementation einer Abhängigkeit zwischen zwei Input-

Elementen, wie im VB.net-Beispiel verdeutlicht, schlicht unmöglich. Eine solche Abhängigkeit muss unter Benutzung der von `shiny` bereit gestellten Update-Funktionen umgesetzt werden, die für fast jede Art von Input-Element im Angebot sind (siehe Code 3). Es gibt - anders als bei VB.net - kein Möglichkeit, dies als Callback zu implementieren. Stattdessen muss ein neuer Observer implementiert werden, der ganz ähnlich wie die Outputs erst von der Reactive-Engine invalidiert werden muss, bevor er ausgeführt werden kann. Das Invalidieren und die erneute Ausführung von Outputs, Observern und Reactive-Values geschieht nicht unmittelbar nach einer benutzerinduzierten Input-Wertänderung, sondern in nacheinander folgenden Server-Ticks.

Programm-Code 3: DependendInputs/app.R

```
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "A", label = "Input A", value = 15, min = 1, ↙
      ↳ max = 100),
  sliderInput(inputId = "B", label = "Input B", value = 10, min = 1, ↙
      ↳ max = 15),
  textOutput("textOut")
)

server <- function(input, output, session) {
  observe({updateSliderInput(session = session, inputId = "B",
           value = min(input$A, input$B),
           max = input$A)})

  output$textOut <- renderText({
    print(paste0("Input A: ", input$A, " - Input B: ", input$B))
  })
}

shinyApp(ui = ui, server = server)
```

Damit ergibt sich folgende Reihenfolge der Ausführung unter Beibehaltung des Beispiels:

1. Zwischen zwei Ticks: Input A wird unter den gegenwärtigen Wert von B gesenkt.
2. Nächster Tick: Der Output (z. B. ein Plot) sowie der Observer zum Anpassen von Input B werden invalidiert.
3. Im selben Tick: Je nach interner Reihenfolge wird zunächst der Output ausgewertet, oder der Observer. Der Observer hinterlegt eine Änderung des Maximums und des Wert von Input B.
4. Anschließend: Die Auswirkung der Update-Funktion zur Änderung der Input-Elemente wird (im Browser) umgesetzt.
5. Nächster Tick: Der Output wird erneut invalidiert, da sich jetzt der Wert von Input B geändert hat.
6. Im selben Tick: Der Output-Code wird erneut ausgeführt, diesmal mit dem angepassten Wert B.

Entscheidend ist, dass sich im Schritt 3 bei der Ausführung des Output-Codes

der Wert von Input B noch nicht geändert hat - und zwar unabhängig davon, ob der Updater-Observer vor oder nach dem Output-Code zur Ausführung kommt. Dies kann zu unerwünschten Ergebnissen oder gar Fehlermeldungen führen, wenn z. B. verwendete Funktionen voraussetzen, dass er kleiner oder gleich dem Wert von Input A ist.

Dieser Nachteil ließe sich eventuell mit einer komplexen Konstruktion von Reactive-Values umgehen, zu dem Preis, dass der Benutzer nun nicht mehr die Namen der Inputs in seinem Output-Code verwenden könnte, mit denen er sie kreiert hat. Aus diesem Grund habe ich davon Abstand genommen, eine solche Lösung zu implementieren, und verweise den Nutzer stattdessen auf die Gefahren bei der Nutzung der von mir angebotenen Standardlösungen, wenn sie solche Abhängigkeiten verwenden.

2.2.2 Zusätzliche Elementeigenschaften

Die Standardlösungen für User-Interfaces bieten außerdem in aller Regel deutlich mehr Möglichkeiten als Shiny, das Aussehen und die Funktionsweise der UI-Elemente zu steuern. Dazu gehören das Aktivieren/Deaktivieren ohne Ausblenden (häufig indem das Element grau gezeichnet wird und Eingaben nicht mehr möglich sind) oder auch Tooltips, die dem Nutzer Hinweise zu einem Element geben, sobald er mit dem Mauszeiger über das Element streicht (ohne zu Klicken). Die Umsetzung dieser Funktionen in `shiny` ist möglich, aber äußerst umständlich.

Ich sehe es daher als Kern meiner Aufgabe, die Steuerung dieser Funktionen zu vereinfachen - neben der im vorigen Kapitel erwähnten einfacheren Deklarierung von Abhängigkeiten zwischen Input-Elementen. Doch zunächst galt es, die grundsätzlichen Funktionen des Parsens des vom Nutzer eingegebenen Codes zur Erzeugung der shiny-App zu implementieren.

3 Umsetzung

3.1 Grundsatzentscheidungen

3.1.1 Optischer Stil der Applikationen

Das Paket `shiny` bietet eine Reihe von Schemata und Stilen der User-Interfaces für die erzeugten Apps. Schon früh wurde jedoch in Abstimmung mit dem Betreuer entschieden, auf das standardisierte Dashboard von `shiny` zu setzen, das sich mit Hilfe des Zusatzpakets `shinydashboard` erzeugen lässt. Es bietet eine sehr klare Zuordnung von Input-Elementen auf der linken Seite und einem großen Output-Bereich auf der rechten Seite. Auch auf den Hilfe-Seiten von `shiny` wird das `shinydashboard` als zu bevorzugende Fortentwicklung der shiny-apps gepriesen.

3.1.2 Notwendige Zusatzpakete

Es gibt für "R" unerhört viele Zusatzpakete mit Lösungen für alle möglichen Detailprobleme. Beim Recherchieren von Code-Lösungen stößt man immer wieder auf

Empfehlungen anderer Benutzer, doch einfach auf dieses oder jenes kleine Zusatzpaket zurückzugreifen. Ich habe mich bewusst gegen die Einbindung weiterer Pakete entschieden, da mein Ziel eine Funktion war, die möglichst sofort und stand-alone funktionieren sollte, und den Benutzer nicht dazu nötigen soll, weitere Pakete zu installieren, die nur für sehr spezielle Details notwendig sind, und die er womöglich nie wieder anderswo benutzen wird.

Die einzig erforderlichen Pakete sind somit diejenigen, die direkt mit `shiny` verbunden sind, also `shinydashboard` für das Dashboard, `shinyBS` für Tooltips und `shinyjs` für das Deaktivieren von Elementen.

3.1.3 Zugang und Output

Zunächst war nur ein R-Script mit einer Funktionsdefinition geplant. Ich entschied mich jedoch, diese Funktion und alle zusätzlich nötigen Hilfsfunktionen, Templates, etc. in einem Paket-Projekt namens `StatAppCreate` zusammenzufassen. Dadurch entfällt zum einen die Notwendigkeit, die R-Scripte per `source` einzubinden, zum anderen lassen sich die klassischen Hilfedateien erzeugen, wie sie für R-Funktionen üblich sind. Dies war mir wichtig.

Das Paket liegt in unkompilierter Form vor, so dass die R-Scripte für die Prüfer in ihrer ursprünglichen Form einsehbar sind. Idealerweise wird direkt das File `StatAppCreate.Rproj` in R-Studio geöffnet. Mit dem Paket `devtools` und dem darin enthaltenen Befehl `load_all` kann dann das Paket so geladen werden, als hätte es ein Endnutzer ganz normal installiert. Als Working-Directory sollte der Pfad der Projektdatei gesetzt werden. Anschließend stehen alle darin enthaltenen Funktionen und Hilfedateien zur Verfügung.

Kern des Pakets ist die Funktion `createApp`, die nicht nur - gegenenfalls in einem vom Nutzer bestimmbaren Unterverzeichnis - die app-Files erzeugt, sondern bei Bedarf auch gleich die erzeugte App startet.

3.1.4 Servercode

Für den grundsätzlichen Server-Code, also jenen Bestandteil der shiny-App-Definitionen, der die Input-Werte verarbeitet und zum Output verarbeitet, biete ich über die bereits erwähnten vereinfachten Abhängigkeiten von Input-Elementen untereinander hinaus keine speziellen Lösungen an. Server-Code ist in der Regel sehr speziell und lässt sich nur schwer schematisieren und kategorisieren. Wie bei jeder Vereinfachung, die ein Parser-Programm bietet, musste auch hier ein Kompromiss zwischen der Vereinfachung und dem damit verbundenen Verlust an Flexibilität für den Anwender gefunden werden.

Im Fall des Server-Codes wogen die Vorteile der Vereinfachung den Verlust an Flexibilität bei weitem nicht auf. Die große Bandbreite möglicher Output-Stile und -Varianten hätte den Parser enorm aufgebläht, ohne einen wirklichen Vorteil einer Vereinfachung zu bieten - schließlich müsste der Anwender dafür die neu eingeführte Syntax des Parsers ebenfalls erlernen. Wenn sich jedoch Parser- und Original-Syntax

in ihrer Komplexität kaum unterscheiden, stellte dies eine völlig unnötige zusätzliche Hürde dar.

3.2 Funktionsweise

Eine ausführliche Beschreibung aller angebotenen Funktionen, ihrer Argumente und detaillierte Beschreibungen der Wirkung sowie mögliche Anwendungsbeispiele findet sich gebotenerweise in den Hilfedateien des Paketes, die in zusammengefasster Form im Anhang 6.2 zu finden sind. Daher wird in diesem Kapitel nur auf die grundsätzliche Funktionsweise eingegangen und jedes Feature dem Wesen nach beschrieben.

3.2.1 Hauptfunktion createApp

Alle Informationen, die zur Erzeugung der App benötigt werden, werden als Argumente in die Funktion `createApp` gefüttert. Der Server-Code wird, wie in Kap. 3.1.4 erwähnt, direkt an ein Argument namens `serverCode` übergeben.

Die UI-Elemente können in Form beliebig vieler, namenloser Argumente an die Funktion übergeben werden, und zwar in genau dem Format, wie sie jedem Anwender vertraut sein dürften, der bereits erste Schritte mit `shiny` unternommen hat (s. Code 4). Innerhalb der Funktion werden diese dann von einem so genannten Ellipsis-Argument repräsentiert, das in R-Funktionen den Namen "..." hat (daher auch der alternative Name Three-Dots-Argument).

Programm-Code 4: createAppSimple.R

```
# very simple example

createApp(targetPath = NULL, title = "Example", launchApp = TRUE,
          sliderInput(inputId = "num", label = "Choose a number", ↙
              ↳ value = 50, min = 1, max = 100),
          checkboxInput(inputId = "unif", label = "Uniform", value = ↙
              ↳ FALSE),
          plotOutput("myPlot"),
          serverCode = quote({
            data <- reactive({
              if (input$unif) { runif(input$num) } else { ↙
                  ↳ rnorm(input$num) }
            })
            output$myPlot <- renderPlot({
              hist(data())
            })
          }) )
```

Die `createApp` übernimmt dann die Aufgabe, dieses Ellipsis-Argument, das im Grunde eine Liste ist, zu analysieren. Die einzelnen Teile der Liste können einfache shiny-UI-Elemente sein (auch verschachtelt), sie können ihrerseits Listen solcher Elemente sein, oder es kann sich um den Inhalt eines benamten R-Objektes handeln. Hieraus ergibt sich allerdings eine Herausforderung, die bei der Implementierung ihrer Lösung

in eine Kaskade neuer Herausforderungen mündete, um allen denkbaren Varianten der Argument-Übergabe gerecht werden zu können.

Wenn Input-Elemente für ein shiny-UI deklariert werden, so geschieht das in Form von Funktionsaufrufen wie z. B. `sliderInput(...)`. Werden diese Funktionsaufrufe nun als Argumente an eine weitere Funktion übergeben, so handelt es sich zunächst einmal nicht mehr um Funktionen, sondern um die jeweiligen Rückgabewerte der Funktionen. In diesem Fall sind das so genannte shiny-tags, die bereits fertig konvertierten HTML-Code darstellen. Als solcher ist das UI-Element aber nicht mehr brauchbar für die Erzeugung eines App-Scripts, das ja erst bei seiner Ausführung den eigentlichen HTML-Code erzeugen soll.

Es musste also ein Weg gefunden werden, den eigentlichen Aufruf analysieren zu können. Glücklicherweise bietet "R" innerhalb von Funktionen die Möglichkeit, den eigentlichen Aufruf (Call) in eine Liste zu konvertieren. Die gleichzeitige Analyse der übergebenen Argument-Werte und der ihnen zugrundeliegenden Calls bot nun den Ansatzpunkt zur korrekten Interpretation des Benutzerwunsches.

Eine Ausnahme bildet die Variante, in der shiny-Elemente zunächst mit shiny-Funktionen erzeugt werden, diese einem R-Objekt zugewiesen werden und schließlich `createApp` mit diesem R-Objekt als Argument aufgerufen wird (s. Code 5). Der Call beinhaltet dann nur noch den Namen des R-Objektes, aber der ursprünglich ausgeführte Code, um ihm einen Wert zuzuweisen, ist nicht mehr erhebbar. Dennoch werden solche Objekte von `createApp` nicht ignoriert, sondern als purer HTML-Code in das App-Script eingebunden. Es kann leider nicht garantiert werden, dass die Elemente dann in jedem Fall korrekt angezeigt werden, oder in der erwünschten Form mit den anderen Elementen zusammen arbeiten.

Eine bessere Lösung bietet dafür die Wrapper-Funktion `uiElement`, die in Kap. 3.2.3 näher beschrieben wird. Sie liefert Listen von Calls für shiny-Elemente zurück, die direkt und ohne Missverständnisse von `createApp` interpretiert werden können.

Programm-Code 5: createAppNamedObj.R

```
slider1 <- sliderInput(inputId = "num", label = "Choose a number", ↙
    ↳ value = 50, min = 1, max = 100) # won't work
chkbox1 <- uiElement(checkboxInput(inputId = "unif", label = ↙
    ↳ "Uniform", value = FALSE)) # will work fine

createApp(targetPath = NULL, title = "Example", launchApp = F,
          slider1,
          chkbox1,
          plotOutput("myPlot"),
          serverCode = quote({
            output$myPlot <- renderPlot({
              data <- reactive({
                if (input$unif) { runif(input$num) } else { ↙
                    ↳ rnorm(input$num) }
              })
              output$myPlot <- renderPlot({
                hist(data())
              })
            })
          }) )
```

`createApp` kennt eine Liste von gültigen shiny-Input-Funktionen, und kann so eine Zuordnung der zu erzeugenden UI-Elemente zur Dashboard-Sidebar bzw. zum Dashboard-Ouput-Body vornehmen, je nachdem, ob es sich um ein Input- oder ein Output-Element handelt.

Die Funktion verwendet vordefinierte Templates zur Erzeugung des App-Codes. Die zu erzeugenden Elemente und Code-Fragmente werden an geeignete, klar identifizierbare Stellen in die Templates eingefügt. Der Nutzer kann bestimmen, in welcher der für `shiny` zulässigen Formen die resultierenden Dateien abgespeichert werden (eine Datei namens `app.R`, zwei Dateien `ui.R` und `server.R`, oder drei Dateien `ui.R`, `server.R` und `global.R`). Die jeweiligen Bestandteile des App-Scripts werden automatisch der korrekten Datei zugeordnet.

Zum Starten der App kann entweder das dafür vorgesehene Flag `launchApp` unter den Parametern auf `TRUE` gesetzt werden, oder eine der erzeugten App-Dateien wird geöffnet und die App dort heraus mit R-Studio gestartet.

3.2.2 Input-Parameter-Relationen

Die bereits erwähnten Beziehungen zwischen Input-Elementen werden in Form von Input-Parameter-Relationen dargestellt, die ich künftig stets unter dieser Bezeichnung referenzieren möchte. Alle Input-Funktionen verfügen über eine Reihe von Parametern, auf die andere Inputs einen Einfluss ausüben können (s. Code 3 in Kap. 2.2.1).

Die Definitionen dieser Relationen werden ebenfalls in beliebiger Zahl in die Funktion gefüttert, mit dem Unterschied zu den UI-Elementen, dass der Nutzer diesen Argumenten Namen gibt, die gleichzeitig zur Identifikation des adressierten Input-Elements und des zu manipulierenden Input-Parameters dienen. Sie folgen daher grundsätzlich

der Form:

```
inputId.param = codeArgument
```

Der Punkt dient der Trennung zwischen Input-ID und Parameter-Name des betroffenen Inputs. Der Code sollte server-seitig ausführbarer Code sein, der von anderen Input-Elementen abhängig ist. Ist dies nicht der Fall, gibt `createApp` eine Warnung aus, da der Code möglicherweise nie zur Ausführung kommt (da `shiny` ihn sonst nie invalidiert).

Programm-Code 6: createAppDependend.R

```
# example how to connect different input elements with each other

createApp(targetPath = NULL, title = "Example", launchApp = TRUE,
          sliderInput(inputId = "A", label = "Input A", value = 15, ↙
              ↳ min = 1, max = 100),
          sliderInput(inputId = "B", label = "Input B", value = 10, ↙
              ↳ min = 1, max = 15),
          textOutput("textOut"),

          B.value = quote({min(input$A, input$B)}),
          B.max = quote({input$A}),

          serverCode = quote({
            output$textOut <- renderText({
              print(paste0("Input A: ", input$A, " - Input B: ", ↙
                  ↳ input$B))
            })
          }) )
```

Das Programm erkennt außerdem auch automatisch, ob die angegebenen Parameter gültige, updatebare Parameter des betroffenen Inputs darstellen. Darüberhinaus existieren zwei spezielle Parameter, denen keine Parameter der shiny-Updatefunktionen entsprechen, und die eine Vereinfachung für den Benutzer darstellen sollen: `.tooltip` für das Setzen von Tooltips, und `.enabled` für das Aktivieren bzw. Deaktivieren des Elements. Der Wert für den Tooltip darf dabei nicht wie die anderen Parameter von anderen Input-Elementen abhängig sein, sondern muss eine feststehende Zeichenkette sein.

3.2.3 Wrapperfunktion uiElement

Die Funktion `uiElement` stellt eine Wrapper-Funktion für Argumente dar, die als UI-Elemente in `createApp` gefüttert werden sollen, um die in Kap. 3.2.1 erwähnten Probleme zu umgehen und Teile der UI-Calls in R-Objekten zwischenspeichern zu können. Es ist möglich, beliebig viele `uiElement`-Calls ineinander zu verschachteln. Das Ellipsis-Argument der Funktion kann dieselbe Art von Argumenten verarbeiten wie `createApp`, also auch Input-Parameter-Relationen.

Eine Besonderheit bietet der Parameter `count`. Wenn er auf einen ganzzahligen Wert größer als eins gesetzt wird, so werden die deklarierten UI-Elemente in dieser Anzahl

generiert, wobei in den Zeichenketten-Parametern der shiny-Element-Funktionen nach einem ersetzbaren Index-String gesucht wird, an dessen Stelle die laufenden Nummer gesetzt wird. Ich habe bei diversen Implementationen von shiny-Apps häufig ein und dieselben Elemente mehrmals einfügen wollen, jeweils nur durch eine laufende Nummer unterschieden. Dieses Feature von `uiElement` vereinfacht das Anlegen solcher multiplen Elemente. Auch bei angegebenen Input-Parameter-Relationen wird der Index-String sowohl in den InputID als auch im dazugehörigen Code-Fragment ersetzt (s. Code 7 und den dazugehörigen Output 8 für ein Beispiel).

Programm-Code 7: uiElementCount.R

```
createApp(targetPath = NULL, title = "Example", launchApp = TRUE,
          uiElement(count=5, indexString = "xxx",
          numericInput(inputId = "valuexxx", label = "Value #xxx", ↙
              ↳ value = "xxx"),
          sliderInput(inputId = "weightxxx", label = "Weight #xxx", ↙
              ↳ value = 0.0, min = 0, max = 1, step = 0.05)),
          textOutput("textOut"),
          serverCode = quote({
            output$textOut <- renderText({
              values <- sapply(1:5, function(i) ↙
                  ↳ as.character(input[[paste0("value", i)]]))
              weights <- sapply(1:5, function(i) ↙
                  ↳ as.character(input[[paste0("weight", i)]]))
              paste0("Values: ", paste(values, collapse = " "), " - ↙
                  ↳ Weights: ", paste(weights, collapse = " "))
            })
          }) )
```

Programm-Code 8: uiElementCount.R - Output

```
      numericInput(inputId = "value1", label = "Value #1", value = ↙
          ↳ "1"),
      sliderInput(inputId = "weight1", label = "Weight #1", value = ↙
          ↳ 0,     min = 0, max = 1, step = 0.05),
      numericInput(inputId = "value2", label = "Value #2", value = ↙
          ↳ "2"),
      sliderInput(inputId = "weight2", label = "Weight #2", value = ↙
          ↳ 0,     min = 0, max = 1, step = 0.05),
      numericInput(inputId = "value3", label = "Value #3", value = ↙
          ↳ "3"),
      sliderInput(inputId = "weight3", label = "Weight #3", value = ↙
          ↳ 0,     min = 0, max = 1, step = 0.05),
      numericInput(inputId = "value4", label = "Value #4", value = ↙
          ↳ "4"),
      sliderInput(inputId = "weight4", label = "Weight #4", value = ↙
          ↳ 0,     min = 0, max = 1, step = 0.05),
      numericInput(inputId = "value5", label = "Value #5", value = ↙
          ↳ "5"),
      sliderInput(inputId = "weight5", label = "Weight #5", value = ↙
          ↳ 0,     min = 0, max = 1, step = 0.05)
```

3.3 Replizierte Statistik-Applikationen

Im Folgenden soll ein Überblick über die bereits mit Hilfe von `createApp` replizierten Statistik-Applikationen aus dem Pool der in **Härdle, Klinke und Rönz (2015)**[1] vorgestellten interaktiven Elemente verschaffen werden. Wie aus den bisherigen Erörterungen hervorgeht, wurde dabei jeweils der Fokus auf die Vereinfachung der Input-Element-Deklarationen und ein Angebot an standardisierten Input-Funktionen gelegt, während der Server-Code nahezu identisch sein sollte. Zu diesem Zweck wurde teilweise Programmcode der interaktiven Elemente des Buches, der zur Verfügung gestellt wurde, unter leichten Änderungen wiederverwendet. Die entsprechenden Code-Stellen sind durch Kommentare gekennzeichnet.

Es wurden zwei interaktiven Elemente ausgewählt, wobei das Augenmerk auf eventuelle Besonderheiten seiner Input-Elemente gelegt wurde. Die jeweiligen Programmcodes finden sich im Anhang 6.1.3.

3.3.1 Dotplot

Das interaktive Element des Erzeugens von Dotplots diverser Daten (s. **Härdle, Klinke und Rönz (2015)** [1, S. 62f] ist ein gutes Beispiel für die Fähigkeit des vorliegenden Pakets, auch zusätzliche Dateien verarbeiten zu können. In diesem Fall wurde die Anzahl der verwendeten Rohdaten auf das "cars"-Datenset reduziert, um die prinzipielle Funktionsweise verdeutlichen zu können. Das Setzen des DataSets wird im globalen Kontext vorgenommen. Zudem wird in diesem Beispiel ein `uiOutput` verwendet, der problemlos von `createApp` verarbeitet werden kann.

3.3.2 Verteilungsmodelle

Bei den Verteilungsmodellen wurde die hypergeometrische Verteilung ausgewählt, die eine aus einer Auswahl von Elementen ohne Zurücklegen zurückgehende Verteilung darstellt (s. **Härdle, Klinke und Rönz (2015)** [1, S. 163ff]. Sie ist ein gutes Beispiel für untereinander abhängige Inputs, da die Werte für die Anzahl der Elemente mit dem interessierenden Merkmal und die Zahl der gezogenen Elemente nicht größer sein darf als die Gesamtzahl an Elementen.

4 Fazit und Ausblick

Das vorliegende Programmpaket bietet umfangreiche Möglichkeiten der vereinfachten Erzeugung von Statistik-Applikationen. Dennoch blieben viele Ideen unberücksichtigt bzw. konnten nicht mehr umgesetzt werden. Gern hätte ich zum Beispiel das Feature von `uiElement` ausgebaut, gleich mehrere ähnliche Elemente auf einen Schlag anlegen zu können, z. B. um die Möglichkeit, die Anzahl dynamisch zu gestalten. Dafür wäre jedoch die Benutzung eines dynamischen UIs nötig, durch `uiOutput`, worauf ich zunächst verzichten wollte. Dynamisch veränderbare Tooltips wären ein weiteres schönes Feature.

Der ursprüngliche Ansatz, eine direktere, kontrollierbarere Möglichkeit zu schaffen, um den Signalfluss der Input-Elemente zu steuern, konnte umgesetzt werden. In den Deklarationen der Input-Parameter-Relationen liegt die Stärke der vorliegenden Lösung.

Offen blieb hingegen die Frage, ob nicht im Bereich der Outputs noch vereinfachende Features hätten angeboten werden können. Besonders schmerzt mich dabei, dass ich das Problem der um einen Server-Tick verzögerten Anpassung von Input-Werten (s. Kap 2.2.1) nicht lösen konnte, so dass hier vom Anwender besonderes Augenmerk abgefordert wird, was ich ihm gern erspart hätte. Diese Features wären ein Thema bei einer Erweiterung des Pakets über den Rahmen dieser Arbeit hinaus, ebenso wie die Implementation von Sprach-Einstellungen für die Standardelemente.

Ein weiteres Manko ergab sich aus der unzureichenden Testphase, insbesondere mit anderen Benutzern als dem Autor selbst. Meine Erfahrung als Programmierer sagt mir, dass die Mehrzahl der auftauchenden Fehler nicht vorhergesehen werden kann. Es ist davon auszugehen, dass noch sehr viel mehr Fälle abgefangen werden müssten, die aus möglicherweise fehlerhaften Eingaben resultieren können, um dem Benutzer in jeder Situation sinnvolle Hinweise zur Behebung geben zu können.

Bevor das Paket in dieser Richtung weiterentwickelt würde, sollte jedoch über alternative Ansätze nachgedacht werden. Da wäre zum einen die Implementierung nicht als reine Funktionen, sondern durch das Anlegen von Klassen. Viele in dieser Arbeit erwähnte Probleme wie zB die Übergabe von shiny-tags an Stelle von UI-Definitionen könnten somit leichter gelöst werden. Zudem bietet der objekt-orientierte Ansatz von R-Klassen viel mehr Möglichkeiten zur exakt abgestimmten Behandlung von Problemfällen. Allerdings wäre für die Umsetzung mit Klassen erst einmal ein umfangreicher, robuster Rohbau anzufertigen, der sich nicht im Rahmen einer Bachelorarbeit hätte darstellen lassen.

Ein noch umfangreicherer, aber auch umso vielversprechender Ansatz wäre es, die Implementation gleich auf der shiny-Ebene vorzunehmen, also eine Erweiterung für shiny selbst zu entwerfen. Auf diese Weise stünden der Manipulation dessen, was der übliche shiny-Output ist, keinerlei Grenzen mehr entgegen.

Die Zahl der replizierten interaktiven Elemente (s. Kap. 3.3) kann nicht zufrieden stellen, auch vor dem Hintergrund der Aufgabenstellung. Gern hätte ich gezeigt, dass die vorliegende Lösung nicht nur auf dem Papier sämtliche der Elemente replizieren kann, sondern auch den praktischen Beweis vorgelegt. Letztlich kann ich nur auf unerwartete Schwierigkeiten bei der Implementation der Hauptfunktion und den daraus resultierenden Zeitmangel verweisen.

Dennoch verbleibe ich in der Hoffnung, dass die Mächtigkeit der hier angebotenen Lösung und der Aufwand ihrer Implementation überzeugen kann. Sehr gerne würde ich mich an der Weiterentwicklung beteiligen.

5 Literaturverzeichnis

Literatur

[1] W. Härdle, S. Klinke, and B. Rönz. *Introduction to Statistics. Using Interactive MM*Stat Elements*. Springer International Publishing, 2015.

6 Anhang

6.1 Programmdateien

6.1.1 Hauptfunktion

Programm-Code 9: createApp.R

```
createApp <- function(...,
                      serverCode = quote({}),
                      globalCode = quote({}),
                      templates = "templates",
                      title = NULL,
                      targetPath = title,
                      headerTitle = title,
                      libraries = c(),
                      includes = c(),
                      additionalFiles = c(),
                      splitUiAndServer = FALSE,
                      includeGlobalR = FALSE,
                      launchApp = FALSE
                      ) {

  ######################
  ### initialization ###
  ######################

  globalTemplateFile <- "tmpl_global.R"
  uiTemplateFile <- ifelse(splitUiAndServer, "tmpl_splitfiles_ui.R", ↙
      ↳ "tmpl_1file_ui.R")
  serverTemplateFile <- ifelse(splitUiAndServer, ↙
      ↳ "tmpl_splitfiles_server.R", "tmpl_1file_server.R")

  updateableShinyElements <- c('actionButton', 'checkboxInput', ↙
      ↳ 'checkboxGroupInput', 'dateInput', 'dateRangeInput',
                          'numericInput', 'radioButtons', ↙
                              ↳ 'selectInput', 'sliderInput', ↙
                              ↳ 'textInput', 'textAreaInput')
  updateFunctions <- paste0("update", ↙
      ↳ toupper(substr(updateableShinyElements, 1, 1)), ↙
      ↳ substring(updateableShinyElements, 2))
  allShinyInputElements <- c(updateableShinyElements, 'actionLink', ↙
      ↳ 'submitButton', 'fileInput', 'passwordInput')
  standardAttributes <- c('class', 'comment', 'dim', 'dimnames', ↙
      ↳ 'names', 'row.names')

  ## load the basic templates
  conn <- file(paste0(templates, "/", globalTemplateFile), open = "r")
  globalTemplate <- paste(readLines(conn), collapse = "\n")
  close(conn)
  conn <- file(paste0(templates, "/", uiTemplateFile), open = "r")
  uiTemplate <- paste(readLines(conn), collapse = "\n")
```

```
36  close(conn)
37  conn <- file(paste0(templates, "/", serverTemplateFile), open = "r")
38  serverTemplate <- paste(readLines(conn), collapse = "\n")
39  close(conn)
40
41
42
43  # analyze ellipsis argument, consisting of calls and input-param relations
44  argValues <- list(...)
45  argCalls <- as.list(substitute(list(...)))[-1]
46  argNames <- names(argValues)
47  dups <- anyDuplicated(argNames[which(argNames != "")])
48  if ( dups > 0 ) {
49    stop(paste0("no unique definition for argument ", argNames[which(argNames != "")][[dups]]))
50  }
51  sidebarCalls <- list()
52  bodyCalls <- list()
53  relations <- list()
54  updateableElements <- list()
55  allInputs <- list()
56  for (i in 1:length(argValues)) {
57    val <- argValues[[i]]
58    asCall <- NULL
59    if ( ! is.language(val) && ! is.character(val) ) {
60      # recognize results of shiny functions
61      if ( is(val)[[1]] == "shiny.tag" ) {
62        # try to retrieve the call that caused the shiny tag object, and use it instead of the produced shiny.tag
63        if ( is.call(argCalls[[i]]) ) {
64          asCall <- argCalls[[i]]
65        } else {
66          # if not accessible (e.g. because stored in an R object), wrap it with the HTML-command of shiny
67          if ( length(grep("input", as.character(val), ignore.case = TRUE)) > 0 ) {
68            sidebarCalls <- c(sidebarCalls, call("HTML", as.character(val)))
69          } else {
70            bodyCalls <- c(sidebarCalls, call("HTML", as.character(val)))
71          }
72        }
73      } else if ( is.list(val) ) {
74        # recognize lists of calls - they will be taken care of in the next step
75        asCall <- val
76      }
77    } else {
78      if ( argNames[[i]] != "" ) {
79       # recognize input-param relations, which should always have a value which is a piece of code
```

```
80          relations[[argNames[[i]]]] <- val
81        } else if ( is.call(argValues[[i]]) ) {
82          # by standard, the value is just a call and can be used ↙
                ↳ right-away
83          asCall <- argValues[[i]]
84        }
85      }
86      if ( ! is.null(asCall) ) {
87        # recognize lists of calls, alike those resulting from helper ↙
              ↳ functions like uiElement
88        if ( length(asCall) > 1 && sum(sapply(asCall, is.call)) != ↙
              ↳ length(asCall) ) {
89          if ( is.language(asCall) && ! is.list(asCall) ) {
90            # if not a list object already, convert the calls into a ↙
                  ↳ list
91            asCall <- list(asCall)
92          }
93        }
94        for (j in 1:length(asCall)) {
95          # handle call by call, and take over the attributes if not ↙
                ↳ already set
96          cll <- asCall[[j]]
97          attribs <- attributes(asCall)
98          for (attrName in names(attributes(cll))) {
99            if ( ! attrName %in% names(attribs) ) {
100             attribs[[attrName]] <- attr(newCall, cll)
101           }
102         }
103
104         # recursively check for nested calls of helper functions ↙
                ↳ like uiElement that still need evaluation
105         cll <- resolveNestedCalls(cll)
106
107         # where to place the ui element resulting from the call
108         side <- length(grep("input", all.names(cll), ignore.case = ↙
                ↳ TRUE)) > 0 || "uiOutput" %in% all.names(cll)
109
110         # retrieve attributes of the call, which might be more ↙
                ↳ input-param relations, or optional parameters
111         for (attrName in names(attribs)) {
112           if ( attrName == "inSidebar") {
113             # eventually override ui element location
114             side <- attribs[[attrName]]
115           } else if ( ! attrName %in% standardAttributes ) {
116             if ( is.character(attribs[[attrName]]) || ↙
                    ↳ is.call(attribs[[attrName]])) {
117               relations[[attrName]] <- attribs[[attrName]]
118             }
119           }
120         }
121
122         # identify shiny input elements for later to allow ↙
                ↳ connection to update code fragments
```

```
123         cllAsText <- paste(deparse(cll), collapse = "")
124         for (elem in allShinyInputElements) {
125           indices <- gregexpr(elem, cllAsText)
126           for (index in indices[[1]]) {
127             if ( index != -1 ) {
128               remain <- substring(cllAsText, index)
129               nextInputId <- regexpr("inputId[\t ]*=[\t ]*\\\"", ↙
                      ↳ remain)
130               remain <- substring(remain, nextInputId + ↙
                      ↳ attr(nextInputId, "match.length"))
131               nextInputId <- regexpr("[A-Za-z.]+[A-Za-z._0-9]*", ↙
                      ↳ remain)
132               inputIdString <- substring(remain, nextInputId, ↙
                      ↳ nextInputId + attr(nextInputId, "match.length")-1)
133               allInputs[length(allInputs)+1] <- inputIdString
134               if ( elem %in% updateableShinyElements ) {
135                 updateableElements[[length(updateableElements)+1]] ↙
                        ↳ <- list(el = elem, nm = inputIdString, up = ↙
                        ↳ updateFunctions[[which(updateableShinyElements==elem)]])
136               }
137             }
138           }
139         }
140
141         # add the ui element to the correct stack of elements
142         if ( side ) {
143           sidebarCalls <- c(sidebarCalls, cll)
144         } else {
145           bodyCalls <- c(bodyCalls, cll)
146         }
147       }
148     }
149   }
150
151   ########################################
152   ### extend the templates step by step ###
153   ########################################
154
155   ########################################
156   # collect server code fragments ...
157   serverCodeAsText <- serverTemplate
158   insertIndex <- regexpr("print\\(1\\)", serverCodeAsText, useBytes ↙
        ↳ = F)
159   lastLineBreakAt <- regexpr("\n[\t ]*$", ↙
        ↳ substring(serverCodeAsText, 1, insertIndex-1), useBytes = F)
160   indentString <- substring(serverCodeAsText, lastLineBreakAt+1, ↙
        ↳ insertIndex-1)
161
162   # handle server code argument
163   serverCodeStr <- NULL
164   if ( is.list(serverCode) || is.vector(serverCode) && ! ↙
        ↳ is.character(serverCode) ) {
```

```
165     stop("invalid argument \'serverCode\': expected character or ↙
            ↳ call")
166   } else if ( is.character(serverCode) ) {
167     serverCodeStr <- serverCode
168   } else if ( class(serverCode)=="{" ) {
169     serverCodeStr <- paste(as.character(serverCode)[-1], collapse = ↙
            ↳ paste0("\n", indentString, "\t"))
170   } else if ( is.call(serverCode) ) {
171     serverCodeStr <- deparse(serverCode)
172   }
173
174   # add input-param relations
175   includeBS <- FALSE
176   includeJS <- FALSE
177   tooltipStr <- NULL
178   relationsStr <- ""
179   for (relName in names(relations)) {
180     # convert relation code to string
181     relVal <- relations[[relName]]
182     relCodeStr <- NULL
183     if ( is.character(relVal) ) {
184       relCodeStr <- relVal
185     } else if ( class(relVal)=="{" ) {
186       relCodeStr <- as.character(relVal)[-1]
187     } else if ( is.call(relVal) ) {
188       relCodeStr <- deparse(relVal)
189     }
190     if ( is.null(relCodeStr) ) {
191       stop(paste0("invalid argument to ", relName, ": expected ↙
              ↳ character or call"))
192     }
193     # retrieve inputId and parameter to manipulate by finding the ↙
            ↳ last . in relation name
194     dotIndex = regexpr(".[a-z]+$", relName)
195     if ( dotIndex == -1 || dotIndex == nchar(relName) ) {
196       stop(paste0("invalid input identifier string ", relName, ": ↙
              ↳ expected inputId followed by . and parameter name"))
197     }
198
199     getsUpdated <- ( sum(sapply(allInputs, function(x) ↙
            ↳ {regexpr(paste0("input\\$", x), relCodeStr) != -1})) > 0 )
200     inputIdStr <- substring(relName, 1, dotIndex-1)
201     inputParamStr <- substring(relName, dotIndex+1)
202     newUpdate <- NULL
203     tooltips <- c()
204
205     if ( ! getsUpdated && inputParamStr != "tooltip" ) {
206       warning(paste0("the code associated with ", relName, " does ↙
              ↳ not contain any known input element IDs and may not get ↙
              ↳ triggered"))
207     }
208
209     if ( inputParamStr == "tooltip" ) {
```

```
210       if ( getsUpdated ) {
211         stop(paste0("dynamic tooltips are not supported yet: ", ↙
              ↳ relName))
212       }
213       includeBS <- TRUE
214       tooltips <- c(tooltips, paste0("bsTooltip(id = \"", ↙
              ↳ inputIdStr, "\", title = \"", relCodeStr, "\")"))
215     } else if ( inputParamStr == "enabled" ) {
216       includeJS <- TRUE
217       newUpdate <- paste0(relName, " <- reactive({", relCodeStr, ↙
              ↳ "})\n",
218                           indentString, "observe({toggleState(id = ↙
                                  ↳ \"", inputIdStr, "\", condition = ", ↙
                                  ↳ relName, "())})\n")
219     } else {
220       updateListIndex <- match(TRUE, sapply(updateableElements, ↙
              ↳ function(x) {x$nm == inputIdStr}), nomatch = 0)
221       if ( updateListIndex == 0 ) {
222         if ( inputIdStr %in% allInputs ) {
223           stop(paste0("invalid input identifier string ", relName, ↙
                  ↳ ": ", inputIdStr, " is not an updateable input ↙
                  ↳ element"))
224         } else {
225           stop(paste0("invalid input identifier string ", relName, ↙
                  ↳ ": ", inputIdStr, " not found among input element ↙
                  ↳ IDs"))
226         }
227       }
228
229       elemToUpdate <- updateableElements[[updateListIndex]]
230       updateableParams <- setdiff(names(formals(elemToUpdate$up)), ↙
              ↳ c('session', 'inputId'))
231       if ( ! inputParamStr %in% updateableParams ) {
232         stop(paste0("invalid input identifier string ", relName, ": ↙
                ↳ ", inputParamStr, " is not an updateable parameter of ↙
                ↳ the input element ", inputIdStr))
233       }
234       # create the observeEvent to trigger the update function
235       newUpdate <- paste0(relName, " <- reactive({", relCodeStr, ↙
              ↳ "})\n",
236                           indentString, "observe({", ↙
                                  ↳ elemToUpdate$up, "(session = ↙
                                  ↳ session, inputId = \"", inputIdStr, ↙
                                  ↳ "\", ", inputParamStr, " = ", ↙
                                  ↳ relName, "())})\n")
237     }
238     if ( ! is.null(newUpdate) ) {
239       relationsStr <- paste0(relationsStr, newUpdate, "\n", ↙
              ↳ indentString)
240     }
241   }
242
243   # finally paste relation strings and server code argument together
```

```
244 serverCodeAsText <- paste0(substring(serverCodeAsText, 1, ↙
        ↳ insertIndex[[1]]-1), relationsStr, serverCodeStr, ↙
        ↳ substring(serverCodeAsText, ↙
        ↳ insertIndex[[1]]+attr(insertIndex, "match.length")))
245
246 ########################################
247 # ui code - all the shiny elements
248
249 uiCodeAsText <- uiTemplate
250 # app title
251 uiCodeAsText <- sub("##TITLE##", title, uiCodeAsText)
252 # header title
253 uiCodeAsText <- sub("##HEADERTITLE##", headerTitle, uiCodeAsText)
254
255 # include input elements into the sidebar
256 inputElems <- ""
257 insertIndex <- regexpr("HTML\\(1\\)", uiCodeAsText, useBytes = F)
258 lastLineBreakAt <- regexpr("\n[\t ]*$", substring(uiCodeAsText, 1, ↙
        ↳ insertIndex-1), useBytes = F)
259 indentString <- substring(uiCodeAsText, lastLineBreakAt+1, ↙
        ↳ insertIndex-1)
260
261 # specific handling if JS is required for disabling/enabling UI ↙
        ↳ elements
262 if ( includeJS ) {
263   inputElems <- paste0(indentString, "useShinyjs()\n")
264 }
265 for (cll in sidebarCalls) {
266   if ( nchar(inputElems) > 0 ) {
267     inputElems <- paste0(inputElems, ",\n", indentString)
268   }
269   nextInputElem <- paste(deparse(cll), collapse = "")
270   nextInputElem <- gsub("\n", paste0("\n", indentString, "\t"), ↙
        ↳ nextInputElem)
271   inputElems <- paste0(inputElems, nextInputElem)
272 }
273 # set tooltips if required
274 if ( includeBS && length(tooltips) > 0 ) {
275   for (tt in tooltips) {
276     inputElems <- paste0(inputElems, ",\n", indentString, tt)
277   }
278 }
279
280 uiCodeAsText <- paste0(substring(uiCodeAsText, 1, insertIndex-1), ↙
        ↳ inputElems, substring(uiCodeAsText, ↙
        ↳ insertIndex+attr(insertIndex, "match.length")))
281
282 # and output elements into body
283 outputElems <- ""
284 insertIndex <- regexpr("HTML\\(2\\)", uiCodeAsText, useBytes = F)
285 lastLineBreakAt <- regexpr("\n[\t ]*$", substring(uiCodeAsText, 1, ↙
        ↳ insertIndex-1), useBytes = F)
```

```
286   indentString <- substring(uiCodeAsText, lastLineBreakAt+1, ↙
          ↳ insertIndex-1)
287   for (cll in bodyCalls) {
288     if ( nchar(outputElems) > 0 ) {
289       outputElems <- paste0(outputElems, ",\n", indentString)
290     }
291     nextOutputElem <- paste(deparse(cll), collapse = "")
292     nextOutputElem <- gsub("\n", paste0("\n", indentString, "\t"), ↙
          ↳ nextOutputElem)
293     outputElems <- paste0(outputElems, nextOutputElem)
294   }
295   uiCodeAsText <- paste0(substring(uiCodeAsText, 1, ↙
          ↳ insertIndex[[1]]-1), outputElems, substring(uiCodeAsText, ↙
          ↳ insertIndex[[1]]+7))
296
297   ########################################
298   # global code - containing libraries and includes
299   globalCodeAsText <- globalTemplate
300
301   # additional libraries
302   if ( includeBS ) {
303     libraries <- c(libraries, "shinyBS")
304   }
305   if ( includeJS ) {
306     libraries <- c(libraries, "shinyjs")
307   }
308   libs <- ""
309   for (lib in rev(libraries)) {
310     libs <- paste(paste0("library(", lib, ")"), libs, sep = "\n")
311   }
312   globalCodeAsText <- sub("##LIBRARIES##", libs, globalCodeAsText)
313
314   # r-scripts to include
315   incs <- ""
316   for (inc in rev(includes)) {
317     incs <- paste(paste0("source(\"", inc, "\")"), incs, sep = "\n")
318   }
319   globalCodeAsText <- sub("##INCLUDES##", incs, globalCodeAsText)
320
321   # handle global code argument
322   globalCodeStr <- NULL
323   if ( is.list(globalCode) || is.vector(globalCode) && ! ↙
          ↳ is.character(globalCode) ) {
324     stop("invalid argument \'globalCode\': expected character or ↙
          ↳ call")
325   } else if ( is.character(globalCode) ) {
326     globalCodeStr <- globalCode
327   } else if ( class(globalCode)=="{" ) {
328     globalCodeStr <- paste(as.character(globalCode)[-1], collapse = ↙
          ↳ paste0("\n", indentString, "\t"))
329   } else if ( is.call(globalCode) ) {
330     globalCodeStr <- deparse(globalCode)
331   }
```

```
332
333   if ( ! is.null(globalCodeStr) ) {
334     globalCodeAsText <- paste0(globalCodeAsText, "\n", ↙
          ↳ globalCodeStr, "\n")
335   }
336
337   ######################
338   ### save the file ###
339   ######################
340
341   if ( !is.null(targetPath) && !is.na(targetPath) && ! ↙
        ↳ dir.exists(targetPath) ) {
342     dir.create(targetPath)
343   }
344
345   pathStub <- ifelse(is.null(targetPath) || is.na(targetPath), "", ↙
        ↳ paste0(targetPath, "/"))
346   if ( splitUiAndServer ) {
347     # ui.R
348     targetFileName <- paste0(pathStub, "ui.R")
349     if ( ! file.exists(targetFileName)) {
350       file.create(targetFileName)
351     }
352     conn <- file(targetFileName, open = "w")
353     writeLines(uiCodeAsText, conn)
354     close(conn)
355
356     # server.R
357     targetFileName <- paste0(pathStub, "server.R")
358     if ( ! file.exists(targetFileName)) {
359       file.create(targetFileName)
360     }
361     conn <- file(targetFileName, open = "w")
362     writeLines(ifelse(includeGlobalR, serverCodeAsText,
363                       paste(globalCodeAsText, serverCodeAsText, sep ↙
                            ↳ = "\n")), conn)
364     close(conn)
365
366     if (includeGlobalR) {
367       # global.R
368       targetFileName <- paste0(pathStub, "global.R")
369       if ( ! file.exists(targetFileName)) {
370         file.create(targetFileName)
371       }
372       conn <- file(targetFileName, open = "w")
373       writeLines(globalCodeAsText, conn)
374       close(conn)
375     }
376   } else {
377     targetFileName <- paste0(pathStub, "app.R")
378     if ( ! file.exists(targetFileName)) {
379       file.create(targetFileName)
380     }
```

```
    conn <- file(targetFileName, open = "w")
    writeLines(paste(globalCodeAsText, uiCodeAsText, ↙
        ↳ serverCodeAsText, sep = "\n"), conn)
    close(conn)
  }

  # copy includes and additionalFiles
  if ( !is.null(targetPath) && !is.na(targetPath) ) {
    for (fl in c(includes, additionalFiles)) {
      if ( dirname(fl) != "." ) {
        path <- unlist(strsplit(dirname(fl), split = "/"))
        for (i in 1:length(path)) {
          newDir <- paste(c(targetPath, path[1:i]), collapse = "/")
          if ( ! dir.exists(newDir) ) {
            dir.create(newDir)
          }
        }
      }
      file.copy(from = fl, to = paste0(targetPath, "/", fl))
    }
  }

  ######################
  ### start the app ###
  ######################

  if ( launchApp ) {
    if ( is.null(targetPath) || is.na(targetPath) ) {
      try(runApp())
    } else {
      oldWD <- getwd()
      setwd(targetPath)
      try(runApp())
      setwd(oldWD)
    }
  }

}

# recursive function to recognize and extrude the helper functions ↙
    ↳ defined in StatAppCreate
resolveNestedCalls <- function(cll) {
  helperFunctions <- c("uiElement", "hyperGeometricPanel", ↙
      ↳ "dotPlotParameterPanel") # ugly hack for now
  cllName <- as.character(cll[[1]])
  attribs <- attributes(cll)
  if ( cllName %in% helperFunctions ) {
    ret <- eval(cll)
  } else {
    argMax <- length(cll)
    newCall <- cll
    offset <- 0
    if ( argMax > 1 ) {
```

```
431       for (i in 2:argMax) {
432         if ( is.call(cll[[i]]) && sum(all.names(cll[[i]]) %in% ↙
              ↳ helperFunctions) > 0 )
433         {
434           replaceCalls <- resolveNestedCalls(cll[[i]])
435           for (attrName in names(attributes(replaceCalls))) {
436             if ( ! attrName %in% names(attribs) ) {
437               attr(newCall, attrName) <- attr(replaceCalls, attrName)
438             }
439           }
440
441           if ( sum(sapply(replaceCalls, is.call)) == ↙
                  ↳ length(replaceCalls) ) {
442             incOffset <- length(replaceCalls) - 1
443             if ( i < argMax ) {
444               for (j in (length(newCall)+ ↙
                      ↳ incOffset):(i+offset+incOffset+1)) {
445                 newCall[[j]] <- newCall[[j-incOffset]]
446               }
447             }
448             for (j in seq(replaceCalls)) {
449               newCall[[i+offset+j-1]] <- replaceCalls[[j]]
450             }
451             offset <- offset + incOffset
452           } else {
453             newCall[[i+offset]] <- replaceCalls
454           }
455         }
456       }
457     }
458     ret <- newCall
459   }
460   for (attrName in names(attribs)) {
461     attr(ret, attrName) <- attribs[[attrName]]
462   }
463   return(ret)
464 }
```

6.1.2 Wrapperfunktion für UI-Elemente

Programm-Code 10: createApp.R

```
1  uiElement <- function(..., inSidebar = TRUE, count = 1, indexString ↙
       ↳ = "index") {
2    argValues <- list(...)
3    argCalls <- as.list(substitute(list(...)))[-1]
4    argNames <- names(argValues)
5    if ( is.null(argNames) ) {
6      argNames <- rep("", length(argCalls))
7    }
8    realCall <- argCalls[which(argNames == "")]
9    relations <- argValues[which(argNames != "")]
10   ret <- NULL
11   if ( count%%1 == 0 && count > 1) {
```

```
12      # if count is larger than one, do some index replacing magic
13      ret <- list()
14      for (i in 1:count) {
15        # recursively replace indexstring in nested ui element calls
16        for (cll in realCall) {
17          ret[[length(ret)+1]] <- substituteIndexInCall(cll, ↙
                ↳ indexString, i)
18        }
19        # replace index string in relation name and code string
20        for (relName in names(relations)) {
21          relVal <- relations[[relName]]
22          relName <- gsub(indexString, as.character(i), relName)
23          relStr <- NULL
24          if ( is.character(relVal) ) {
25            relStr <- relVal
26          } else if ( class(relVal)=="{" ) {
27            relStr <- as.character(relVal)[-1]
28          } else if ( is.call(relVal) ) {
29            relStr <- deparse(relVal)
30          }
31          if ( !is.null(relStr) ) {
32            relStr <- gsub(indexString, as.character(i), relStr)
33            attr(ret, relName) <- relStr
34          }
35        }
36      }
37    } else {
38      # store input relations as attributes of the resulting list of ↙
          ↳ calls, for later interpretation by createApp
39      ret <- realCall
40      for (relName in names(relations)) {
41        attr(ret, relName) <- relations[[relName]]
42      }
43    }
44    if ( ! missing(inSidebar) ) {
45      attr(ret, "inSidebar") <- inSidebar
46    }
47    return(ret)
48  }
49
50  substituteIndexInCall <- function(cll, indexString, index) {
51    adjCll <- as.character(cll)
52    adjCll <- as.list(adjCll)
53    if ( length(adjCll) > 1 ) {
54      for (i in 2:length(adjCll)) {
55        if ( is.call(cll[[i]]) ) {
56          # substitute nested calls recursively
57          adjCll[[i]] <- substituteIndexInCall(cll[[i]], indexString, ↙
                ↳ index)
58        } else if ( is.atomic(cll[[i]]) ) {
59          # substitute index string
60          adjCll[[i]] <- gsub(indexString, as.character(index), ↙
                ↳ adjCll[[i]])
```

```
61          # try evaluating it
62          triedEval <- try(eval(parse(text = adjCll[[i]])), silent = ↙
                ↳ TRUE)
63          if ( class(triedEval) != "try-error" ) {
64            adjCll[[i]] <- triedEval
65          }
66          # restore correct modes of atomic arguments
67          mode(adjCll[[i]]) <- mode(cll[[i]])
68        }
69      }
70    }
71    # restore the names of the arguments
72    names(adjCll) <- names(cll)
73
74    # create the call
75    adjCall <- do.call("call", adjCll, quote = TRUE)
76    return(adjCall)
77  }
```

6.1.3 Standardelemente

Programm-Code 11: HyperGeometricPanel.R (Definition)

```
hyperGeometricPanel <- function(M = 10, n = 20, N = 5) {
  return(uiElement(sliderInput(inputId = "hyper.N", label = ↙
      ↳ "Population Size (N)", value = 20, min = 1, max = 50),
                   sliderInput(inputId = "hyper.M", label = "Number ↙
                       ↳ of success states in the population (M)", ↙
                       ↳ value = 14, min = 1, max = 20),
                   sliderInput(inputId = "hyper.n", label = "Number ↙
                       ↳ of Draws (n)", value = 5, min = 1, max = 20),
         hyper.M.value = quote({min(input$hyper.M, input$hyper.N)}),
         hyper.n.value = quote({min(input$hyper.n, input$hyper.N)}),
         hyper.M.max = quote({input$hyper.N}),
         hyper.n.max = quote({input$hyper.N})
         ))

}
```

Programm-Code 12: DotPlotParameterPanel.R (Definition)

```
dotPlotParameterPanel <- function() {

  return(uiElement(
    selectInput(inputId = "method", label = "Select a Dotplot type", ↙
        ↳ selected = "jitter", choices = c("overplot", "jitter", ↙
        ↳ "stack")),
    checkboxInput(inputId = "addmean", label = "Add mean ↙
        ↳ (aquamarine)", value = FALSE),
    checkboxInput(inputId = "addmedian", label = "Add median ↙
        ↳ (blue)", value = FALSE),
    checkboxInput(inputId = "addrange", label = "Add range ↙
        ↳ (orange)", value = FALSE),
    checkboxInput(inputId = "addiqr", label = "Add interquartile ↙
        ↳ range (pink)", value = FALSE)
  ))

}
```

6.2 Hilfedateien

Package 'StatAppCreate'

January 29, 2017

Title Statistical Application Creation

Version 0.1

Description Provides functions to create shiny apps serving the demonstration of statistical associations

Depends R (>= 3.2.5)

License GPL-2

Imports utils,
shiny (>= 0.13.2),
shinydashboard (>= 0.5.2),
shinyBS (>= 0.61)

RoxygenNote 5.0.1

R topics documented:

`createApp` *Statistical Application Creation*

Description

`createApp` creates files containing R code necessary to launch a shiny app

`uiElement` is a wrapper function for arguments fed into `createApp`

Usage

```
createApp(..., serverCode = quote({ }), globalCode = quote({ }),
  templates = "templates", title = NULL, targetPath = title,
  headerTitle = title, libraries = c(), includes = c(),
  additionalFiles = c(), splitUiAndServer = FALSE, includeGlobalR = FALSE,
  launchApp = FALSE)

uiElement(..., inSidebar = TRUE, count = 1, indexString = "index")
```

Arguments

...	shiny objects or named code chunks. See 'Details' for more information
serverCode	a character or expression representing the basic part of the server code
globalCode	a character or expression representing additional code to be inserted into the global scope
templates	a character. The directory where to find the template files for the apps
title	a character. The title of the app, displayed in the window bar.
targetPath	a character or NULL. The sub directory where to put the app file(s). If missing, will be the same as title. If NULL, the app file(s) will be saved into the current working directory.
headerTitle	a character. The title displayed in the header of the dashboard of the app. If missing, will be the same as title.
libraries	a character vector. Any additional libraries required by the code of the final app should be listed here.
includes	a character vector. Any file that need to be source()'d by the app should be listed here. The files will be copied into the app's directory if necessary, maintaining sub directory structures.
additionalFiles	a character vector. Any non-source file that needs to be in the app's directory should be listed here. The files will be copied into the app's directory if necessary, maintaining sub directory structures.
splitUiAndServer	logical. If FALSE (the default), the app will consist of only one file (app.R), otherwise it will consist of at least two files (ui.R and server.R). See includeGlobalR for additional info.
includeGlobalR	logical. Only relevant if splitUiAndServer is TRUE. If FALSE (the default), the global part of the app's code will be part of server.R), otherwise be a global.R) file of its own.
launchApp	logical. If TRUE, the app will not only be saved, but also launched afterwards.
inSidebar	logical. If TRUE (the default), the created ui elements will be placed in the dashboard's sidebar, otherwise in the body.
count	integer. If larger than 1, the ui element will be created count times, only differing by those parts of the shiny function parameters that are identified by indexString
indexString	a single length character vector. indexString will be used to identify what to replace by a number counting up from 1 to count

Details

createApp tries to interpret its ... argument as calls to create shiny ui elements, processes them and saves them in shiny-style R-files. The result is a shiny-dashboard-app that is working by itself. Any shiny function used to create shiny elements can be used. Elements containing inputs will automatically be placed in the dashboard's sidebar, the remaining ones in the dashboard body. It is possible to use menuItem and menuSubItem. The sidebar's id parameter is 'sidebar'.
Note that createApp is trying to interpret both the call and the return value of the given shiny functions. If shiny function calls are stored in R-objects and used as an argument for createApp they might not get interpreted and parsed correclty. For such cases, use uiElement as a wrapper function.

createApp will try to interpret any named arguments that are not standard named arguments of itself as declarations of input parameter relations. Those are meant to establish dependencies between the value of an input and parameters of another input element. Those declarations always follow the scheme
inputId.parameterName = codeFragment
The codeFragment can be a character vector of length 1, or have the form quote({}) or quote(). Please note that in the latter case the included code will be executed upon calling createApp. It is recommended to use quote({}).
The parameterName can be any parameter of an input element that can be manipulated by the respective shiny update function for that kind of input element.
Other than that, special parameterNames can be used to set the tooltip or the enabled state of the element (see Examples).

All code defining the behavior of output elements will have to be included in the serverCode argument. These code chunks are parsed unchanged. The same is true for the globalCode argument which will be inserted in the global scope of the app scripts.
createApp and the resulting apps do not depend on any packages except the shiny relevant packages such as shiny, shinydashboard, shinyBS and shinyjs.
Any additionally required packages, source()'ed files and other required files need to declared in the respective arguments libraries, includes and additionalFiles.

uiElement takes the same argument types to its ... argument as createApp. This allows to store the result in an R-object for later use with createApp.
The wrapper also provides a possibility to force input elements to appear in the dashboard's body and output elements to appear in the dashboard's sidebar.

If the argument count is given and larger than 1, the ui elements will be added multiple times, only differing by those parts of their inputId's, labels and other string arguments which are identified by indexString and replaced by consequetive numbers (see Examples).
uiElement will try to evaluate the substituted parameter strings.
For input-parameter-relations, their name and code will also be searched for the indexString to substitute it therein.

Examples

```
# very simple example

createApp(targetPath = NULL, title = "Example", launchApp = TRUE,
        sliderInput(inputId = "num", label = "Choose a number", value = 50, min = 1, max = 100),
          checkboxInput(inputId = "unif", label = "Uniform", value = FALSE),
          plotOutput("myPlot"),
          serverCode = quote({
            data <- reactive({
              if (input$unif) { runif(input$num) } else { rnorm(input$num) }
            })
            output$myPlot <- renderPlot({
              hist(data())
            })
          }) )

# example how to connect different input elements with each other

createApp(targetPath = NULL, title = "Example", launchApp = TRUE,
```

Hiermit erkläre ich, Axel Hylla dass ich die vorliegende Arbeit selbständig verfasst, für keine andere Prüfung verwendet und keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet. Ich bin mir darüber bewusst, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, am 29.01.2017, Axel Hylla